Authors: Steven Sarns and Jack Woehr

Embedded Systems™ PROGRAMMING

Exploring I²C

Serial data buses are a well-proven tool in embedded systems. When you are communicating with slow peripheral devices, serial buses are often often more convenient and less expensive than parallel buses. Additionally, a serial interface featuring a UART or similar intermediary chip can also serve to isolate the CPU from noise and line glitches that might bring down the house if they were to occur on the processor bus. Peripherals can usually be controlled over a much greater distance by a serial bus. The serial approach offers greater resilience and noise immunity.

The price you pay for the convenience is a slower transmission rate and, possibly, the need for added interface circuitry at higher voltages. Many peripheral devices, however, are not in constant communication with the CPU and are not greatly affected by a slower bus. On the hardware side, any added interface circuitry required for serial-bus support is frequently compensated for by the resulting simplicity and tighter pinout of the serial peripherals.

The choice of bus and protocol depends at least as much on the system's software as it does on the manufacturer's data sheets.

CHOOSING THE PROPER ROUTE

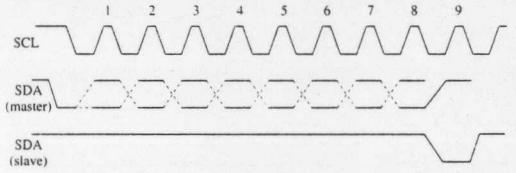
Having decided that a serial bus makes sense for your application, your next task is to select the most appropriate bus and protocol. Here, as with rapid transit, your choice should be determined by your destination. Contrary to what some people may tell you, the choice of bus and protocol depends at least as much on the nature of the system's software as it does on the manufacturer's data sheets.

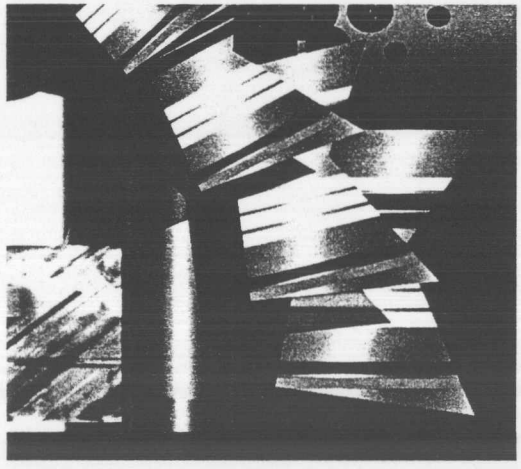
Consider, for example, the serial-peripheral interface (SPI) and multidrop serial buses. Both buses are popular, but each exhibits severly constrained performance in large networks. SPI, as embodied in the Motorola 6800 family, was designed primarily for one-on-one exchanges between two devices. Similarly, the multidrop approach used in various 8051 family members as well as in the 68HC11 and various UART chips finds its broadest expression in RS485/422 half-duplex transmissions. Multidrop has no deterministic arbitration scheme between multiple masters, leaving it mainly suitable for single-master multiple-slave situations. *(For more on multidrop, see Jack Woehr's article, "Multidrop Processing," Embedded Systems Programming, March 1990, pp 58-67—ed.)* A different approach is to use a three-wire protocol called MicroWire, available from National Semiconductor in Santa Clara, Calif., which is fine for use with addressable peripherals, but requires an individual chip select for each device addressed. The added wiring offers no advantage to developers, and the bus offers nothing towards achieving multiple-mastering capabilities.

One of the more versatile options available to developers is the I²C bus promulgated by Philips/Signetics in Sunnyvale, Calif. I²C allows you to set up a multiple-master, multiple-slave communications bus with conflict arbitration, using only twisted-pair wiring to connect the processors and peripherals. Philips/Signetics has moved to support this protocol (which is quite popular in Europe) with a large assortment of interesting doodads, and is actively encouraging other manufacturers to join in the fun. If your next design features a microprocessor that supports I²C or you are prepared to implement I²C in software using a PIA as this article illustrates, your reward could be a decreased chip count and lower power consumption—along with a comfortable distributed-programming model for peripheral devices.

Open-collector configuration means that the output stage can only pull the node to ground.

Figure 1
Generation of acknowledge.





I²C is more flexible than the protocols noted above, since only two wires are required to service a large network of addressable masters and addressable slaves. A third wire may be added if interrupt service is required, though Philips/Signetics microprocessors featuring I²C support feature on-chip circuitry and are capable of interrupting the processor upon receipt of a valid address.

HOW I²C WORKS

The I²C bus consists of two lines: serial clock (SCL) and serial data (SDA). The beauty of the I²C bus is that each of these lines is bidirectional. Bidirectional means that everything on the bus is equal, unlike most other serial-peripheral busses such as SPI or MicroWire, which have dedicated inputs and outputs. Each I²C transaction line (SCL and SDA) is an open collector of output and input. The

pullup resistor is external.

Open-collector (actually, they are CMOS, so "open drain" is more appropriate) configuration means that the output stage can only pull the node to ground. A passive resistor pulls the node high, which means that any number of open collector outputs can be connected together with no deliterious results, because it is impossible to pull more current through the resistor than any one output will produce. Tying outputs together will produce disastrous results if the same procedure is tried with standard TTL outputs. If some of the outputs go high and some are low, the current is unlimited and the logic level of the output will be in an indeterminate state. Tying open-collector outputs together is also known as "wire ORing" because if either A or B goes low, so does the single-output line.

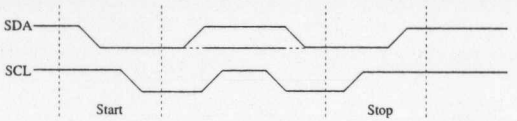
The I²C bus speed is specified at a maximum SCL rate of 100kHz SCL, which, admittedly, is not blazingly fast. The speed limit stems from the meager ability of a pullup resistor to source current to a long distributed line of peripherals. The 10-microsecond period allows plenty of time to charge the parasitic capacitance of the wires. (The maximum specified wire capacitance is 400 pF.)

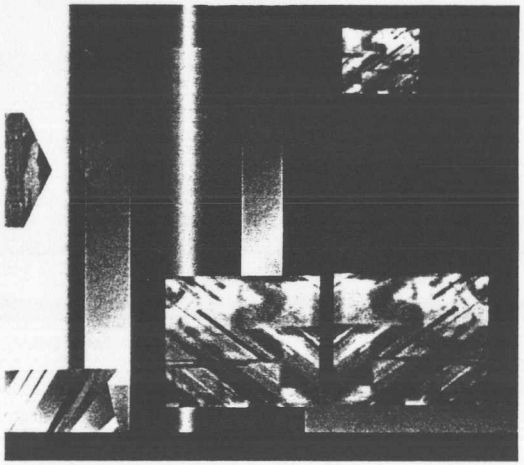
PUTTING IT TOGETHER

Although I²C supports multiple-master operation, here we use single-master, single-slave transactions to keep the example code simple. The master, as you might imagine, is defined as the unit that initiates the data transfer and generates the SCL signal. (In a multimaster system, each master would be responsible for generating its own SCL signal.) In our example, based strongly on the design of one of our company's single-board computers, the processor doesn't directly support I²C. Instead, we've implemented the I²C bus using a couple of the pins on an 8255 peripheral I/O chip. Consequently, the bulk of the example application code is simple setup and housekeeping routines. *(Steven R. Wheeler's example application listing was a bit too long to run in this issue. Interested readers may download it from the library 12 of CLMFORUM on CompuServe or from the* Embedded Systems Programming *bulletin board service at (415) 905-2689—ed.)*

By definition, a slave can be any processor or peripheral that responds to the master. Slaves all have unique, 7-bit addresses that are based on the device type and the wiring of address pins on the chip. All I²C peripherals have the top nibble of an address built in. For the PCF8574 I/O-port expanders we're using as examples, the address is 0100*xxx*. The *xxx* indicates the address selected by the state of the three address pins on the peripheral.

Figure 2
Start and stop conditions.





Exploring I²C

I²C serial transactions are always eight bits of data from the transmitter followed by a ninth ACK bit from the receiver. The first step in any I²C data transfer is to send the address of the slave on the SDA line. This act might seem confusing, since we seem to be mixing 7-bit addresses with 8-bit data. In practice, it's quite easy to work with: addresses are always seven bits long, and the eighth bit is used to determine whether the operation is a read or a write. For example, upon transmitting 01000001 to the PCF8574, the slave, assuming it exists on the bus and is strapped to address 000, will respond with a low on the SDA line after the master has finished with its last (eighth) data bit. The master leaves the line high. If it doesn't find a slave with address 10000, the data line will remain high and a failed communication attempt can be detected.

If a slave is connected, it begins putting data on the SDA line as soon as it has detected that the eighth bit is set (which is a read request). The SDA line is driven to the data level when the SCL line is low. Data is read when SCL is high, so SDA must not change when SCL is high. This protocol leads to a simple definition of the start of an I²C transaction—SDA goes from high to low when the clock is high.

The end of a transaction is equally simple to detect: SDA goes from low to high when SCL is high. This cycle leaves SDA and SCL in the high state, which is necessary if any other open-collector I²C peripheral wants access to the bus. Figure 2 illustrates the start and stop conditions of an I²C bus transaction.

ADDITIONAL DESIGN ROUTES

As you've seen, the I²C protocol is easy to work with and relatively simple to implement, even if you're not using a processor that directly implements it. If you're not planning to use Philips/Signetics microprocessors with onboard I²C support (such as the 68070 or various members of the 8051 family), you can still use the wide variety of available peripheral chips.

The number of integrated circuits using the I²C serial bus is increasing all the time. Application-oriented integrated circuits that support I²C include a voice sythesizer, a transcoder for IR remote control, several digital tuning circuits for computer-controlled television, several audio processors, PLL frequency synthesizers, tone generators, and frequency synthesizers. General-purpose integrated circuits using I²C include LCD drivers, digital-to-analog converters, SRAMs, EEPROMs, and a RAM clock/calender.

I²C is very popular in Europe, where Philips has been aggressively marketing this flexible method of extending peripheral support to control projects, and it is currently catching fire on this side of the Atlantic. It seems reasonable to expect that, given the burden of printed-wire requirements for embedded systems based on increasingly wider chip buses, more and more designers seeking economy of means will be attracted to the economy of I²C.

Steven Sarns is the president of Vesta Technology in Wheat Ridge, Colo. He is a member of Mensa, Intertel, and the Michigan Society of Professional Engineers. Sarns is also a founding member of the Denver chapter of the Forth Interest Group.

Jack Woehr is a senior project manager at Vesta Technology Inc. in Wheat Ridge, Colo. He is a Chapter Coordinator for the Forth Interest Group and is currently a member of the X3J14 Technical committee for ANS Forth. He can be reached by E-mail as jax@well.sf.ca.us or as VESTA on GEnie.